BE A GOOD DEVELOPER

.NET EgM - 2019

NASH TECH

The Power to Innovate

What make a good developer?

What is good code?

"Good code is code that you are proud to release. Code that you know you have done your best on. Code that you "know" works, and that you have given the best design you can think of in the time available. This is not perfection, it is diligence; it is professionalism."

Robert C. Martin

Agenda

- What make a good developer?
- Code smells and how to refactor them
- Code review
- Good practices for developers
- Good books for developers

NASH TECH
The Power to Innovate

Code smells and how to refactor them

What is code smell?

- A hint that something might potentially go wrong in the code
- A hint of coding bad practice
- The cause of complexity of source code
- Makes the code hard to understand and hard to be maintained

Code smells and refactoring

- Refactoring is usually motivated by noticing a code smell
- By doing refactoring on code smells frequently, programmers will be better at programming

Common code smells

- Duplicated Code
- Large class/Super class
- Long Method
- Excessive Indentation (IF Statements)
- Undescriptive Name
- Useless comments

Duplicated code

- The same, or very similar code, appears in many places
- One of the worst code smells
- Makes the code expand and hard to be maintained/controlled

=> Refactor: Extract method

Duplicated code - Example

Before

```
public static XmlElement CreateAddressElement(XmlDocument xmlDocument, Address address)
{
    XmlElement element = xmlDocument.CreateElement("Address");
    XmlAttribute tempAttribute = null;
    tempAttribute = xmlDocument.CreateAttribute("Number");
    tempAttribute.Value = address.Number;
    element.SetAttributeNode(tempAttribute);
    tempAttribute = xmlDocument.CreateAttribute("Street");
    tempAttribute.Value = address.Street;
    element.SetAttributeNode(tempAttribute);
    tempAttribute = xmlDocument.CreateAttribute("City");
    tempAttribute.Value = address.City;
    element.SetAttributeNode(tempAttribute);
    tempAttribute = xmlDocument.CreateAttribute("Country");
    tempAttribute.Value = address.Country;
    element.SetAttributeNode(tempAttribute);
    return element;
}
```

Duplicated code - Example

After

```
public XmlElement CreateAddressElement(XmlDocument xmlDocument, Address address)
{
    XmlElement element = xmlDocument.CreateElement("Address");
    element.SetAttributeNode(CreateAttribute(xmlDocument, "Number", address.Number));
    element.SetAttributeNode(CreateAttribute(xmlDocument, "Street", address.Street));
    element.SetAttributeNode(CreateAttribute(xmlDocument, "City", address.City));
    element.SetAttributeNode(CreateAttribute(xmlDocument, "Country", address.Country));
    return element;
}

private XmlAttribute CreateAttribute(XmlDocument xmlDocument, string name, string value)
{
    XmlAttribute attribute = xmlDocument.CreateAttribute(name);
    attribute.Value = value;
    return attribute;
}
```

Large class/Super class

- A class is trying to do too much, low cohesion
- Methods that don’t interact with the rest of the class
- Fields that are only used by one method
- Classes that change often

=> Refactor: Extract class

Example

Before

Vehicle

- Edit vehicle options
- Update pricing
- Schedule maintenance
- Send maintenance reminder
- Select financing
- Calculate monthly payment

After

Vehicle

- Edit vehicle options
- Update pricing

VehicleMaintenance

- Schedule maintenance
- Send maintenance reminder

VehicleFinance

- Select financing
- Calculate monthly payment

Long method

- White spaces & Comments
- Scrolling required
- Naming issues
- Multiple Conditionals
- Hard to digest

Long method

- How long is too long? There is no official rule
- More than 20 lines should be considered a smell
- Under 10 lines is typically good

=> Refactor: Extract method

Example – Extract method

Before

```
if
    if
        while
            do
            some
            complicated
            thing
        end while
    end if
end if
```

After

```
if
    if
        doComplicatedThing()
    end if
end if


doComplicatedThing()
{
    while
        do some complicated thing
    end while
}
```

Example – Extract method

- Watch for flag arguments (boolean argument) – a sign the function doing two things

```
private void SaveUser(User user, bool emailUser)
{
    //save user

    if (emailUser)
    {
        //email user
    }
}
```

```
private void SaveUser(User user)
{
    //save user
}

private void EmailUser(User user)
{
    //email user
}
```

Excessive Indentation – Complex Conditionals

- Complicated “if” condition or “else if” condition
- Hard to understand and maintain
- Contains potential bugs

=> Refactor: Return early, Intermediate Variables to convey intent and Encapsulate via function

Example – Intermediate variables

Dirty

```
if (employee.Age > 55
    && employee.YearsEmployed > 10
    && employee.IsRetired == true)
{
    //logic here
}
```

Clean

```
bool eligibleForPension = employee.Age > MinRetirementAge
    && employee.YearsEmployed > MinPensionEmploymentYears
    && employee.IsRetired;
```

Example – Return early

```
private bool ValidUsername(string username)
{
    bool isValid = false;

    const int MinUsernameLength = 6;
    if (username.Length >= MinUsernameLength)
    {
        const int MaxUsernameLength = 25;
        if (username.Length <= MaxUsernameLength)
        {
            bool isAlphaNumeric = username.All(Char.IsLetterOrDigit);
            if (isAlphaNumeric)
            {
                if (!ContainsCurseWords(username))
                {
                    isValid = IsUniqueUsername(username);
                }
            }
        }
    }
    return isValid;
}
```

```
private bool ValidUsername(string username)
{
    const int MinUsernameLength = 6;
    if (username.Length < MinUsernameLength) return false;

    const int MaxUsernameLength = 25;
    if (username.Length > MaxUsernameLength) return false;

    bool isAlphaNumeric = username.All(Char.IsLetterOrDigit);
    if (!isAlphaNumeric) return false;

    if (ContainsCurseWords(username)) return false;

    return IsUniqueUsername(username);
}
```

Example - Encapsulate via function

Dirty

```
//Check for valid file extensions. Confirm admin or active
if (fileExtension == "mp4" ||
    fileExtension == "mpg" ||
    fileExtension == "avi")
    && (isAdmin || isActiveFile);
```

Clean

```
if (ValidFileRequest(fileExtension, active))

private bool ValidFileRequest(string fileExtension, bool isActiveFile, bool isAdmin)
{
    var validFileExtensions = new List<string>() { "mp4", "mpg", "avi" };

    bool validFileType = validFileExtensions.Contains(fileExtension);
    bool userIsAllowedToViewFile = isActiveFile || isAdmin;

    return validFileType && userIsAllowedToViewFile;
}
```

Undescriptive name

- Name of method or variable does not reveal its purpose
- Good code has good naming
- Bad naming causes confusion

=> Refactor: Rename

Undescriptive name

- Common convention:
  - Use unabbreviated, correctly-spelled meaningful names
  - Choose a name that describes WHAT the object does, not how it does it
  - Keep names short

Undescriptive name - Example

- Bad:
  - GetAccNo
  - GetSchTask
  - strCompanyName
- Good:
  - GetAccountNumber
  - GetScheduledTask
  - companyName
- Exceptions:
  - min, max, sin, cos, abs…
  - for (i = 0; i < 10; i++)

Comment

- Comment is also a smell when it describes “what” the code does
- Having too many comments will make the code become messy
- The code should be expressive enough for person who reads it understand it
- Use comment to say “why” you did something

Comment - Example

```
public static bool SHA1VerifyString(string input, string hash)
{
    // Hash the input.
    string hashOfInput = StringHelper.SHA1String(input);

    // Create a StringComparer an comare the hashes.
    StringComparer comparer = StringComparer.OrdinalIgnoreCase;

    return 0 == comparer.Compare(hashOfInput, hash);
}
```

Code smells - Summary

- Code smells are signs of coding bad practice
- Eliminating code smells by refactoring helps make your code clean and robust
- Less code smells means less potential issues
- By refactoring code smells frequently, you will become a better programmer
- Make sure you don’t introduce any bug when you refactor code smells

Code review

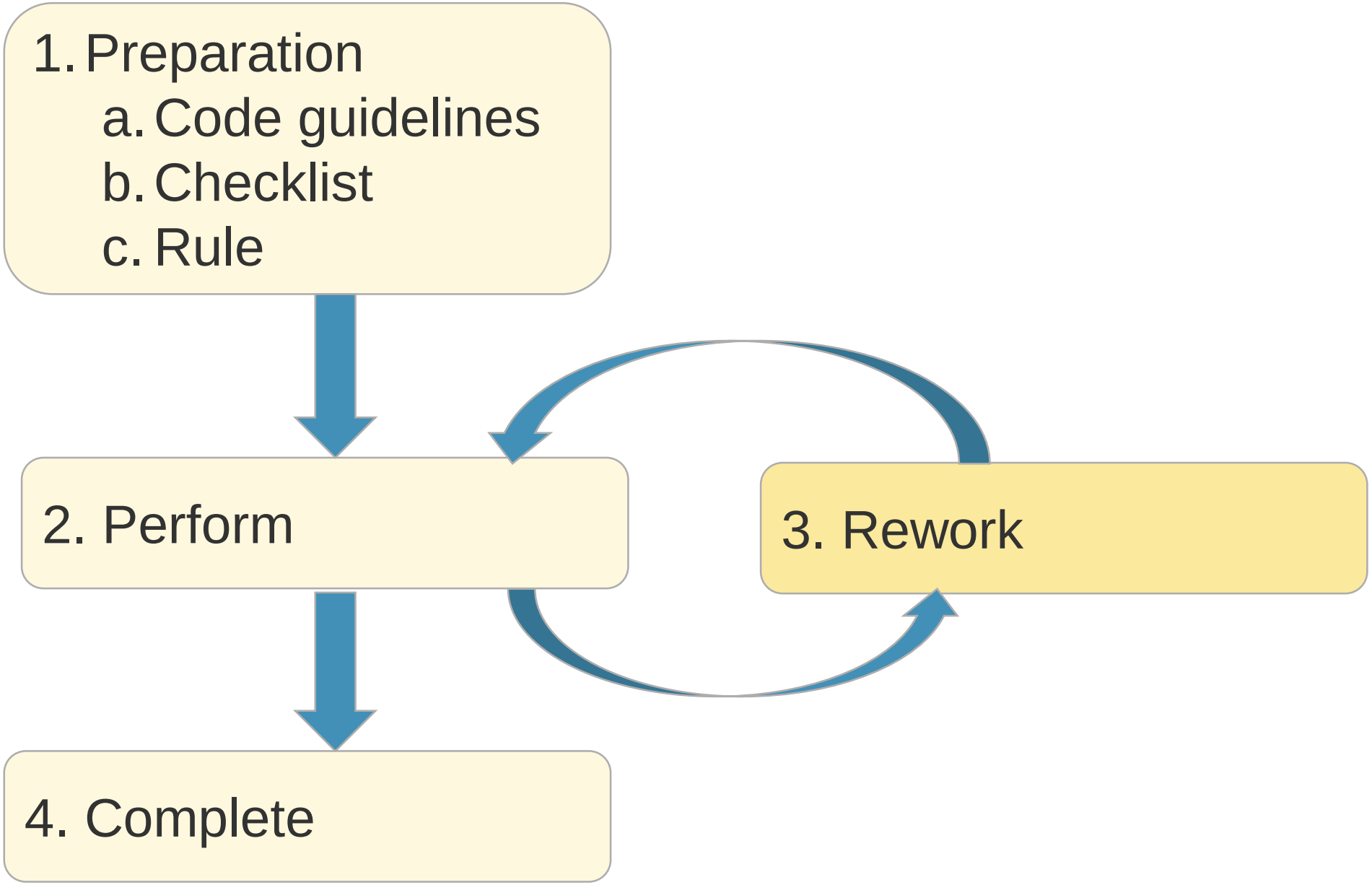
What is code review

- Code review is a phase in the software development process
- It is intended to find and fix mistakes
- Reviewers read the code line by line to check for:
  - Errors or potential errors
  - Consistency with the overall program design
  - The quality of comments
  - Adherence to coding standards

Why review code

- Finding defects - Improve the code quality
- Learning/Knowledge transfer

Code Review Guidelines



Code Guidelines

- Code Guideline should include:
  - File Organization
  - Style guidelines
  - Commenting guidelines
  - Naming Conventions
  - Statements
  - Exception Handling
  - Logging
  - Unit Testing
  - …

Checklist for code review

- Code review checklist should include the following areas but not limited
  - ***Functionality***: Does the code meet the business (functional) requirements? Is the logic correct?
  - ***Class Design***: Does each class have low coupling and high cohesion? Is the class too large or too complicated? Does the code completely and correctly implement the design?
  - ***Code Style***: Is duplication of code avoided? Are any methods too long? Are typical coding idioms / standards followed?
  - ***Naming***: Are packages, classes, methods and fields given meaningful and consistent names?
  - ***Error Handling***: How are errors dealt with? Does the code check for any error conditions that can occur?
  - ***Security***: Does the code require any special permission to execute outside the norm? Does the code contain any security holes?
  - ***Unit Tests***: Are there automated unit tests providing adequate coverage of the code base? Are these unit tests well-written?

Code review rules

- Must have self-review before request review
- Defects are logged

Common mistakes

- **Database design, development**
  - Wrong data type
  - Inconsistent naming
  - Not release/clean up resource well
- **Normal coding**
  - Bad while loop, for
  - Code smell, design smell
  - Not enough validation
  - Security risk
  - Swallow Exception
  - Incorrect async/await usages
- **Performance Issues**
  - Resource cleanup
  - String Management
  - Threading
  - Boxing
  - N-1

NASH TECH
The Power to Innovate

Some good practices for developers

NEVER share client’s or company’s artifacts such as source code, documents to any body without approval from higher manager.

Be One Team

- Code is the code of team not of any individual person
- Follow the team conventions

NEVER Underestimate the Foundation

Be Proactive

Whenever you receive a task

- Try to get all the information and understand it clearly
- Raise, if there are something not clear, concern or uncertain
- Raise, if there is any impediments during the implementation
- NEVER wait until the final day

Be Proactive

- Learn skills according to the competency development plan
- Talk to your Line Manager if you need any support

A developer who relies upon QC to test their code is a bad programmer

Working practices

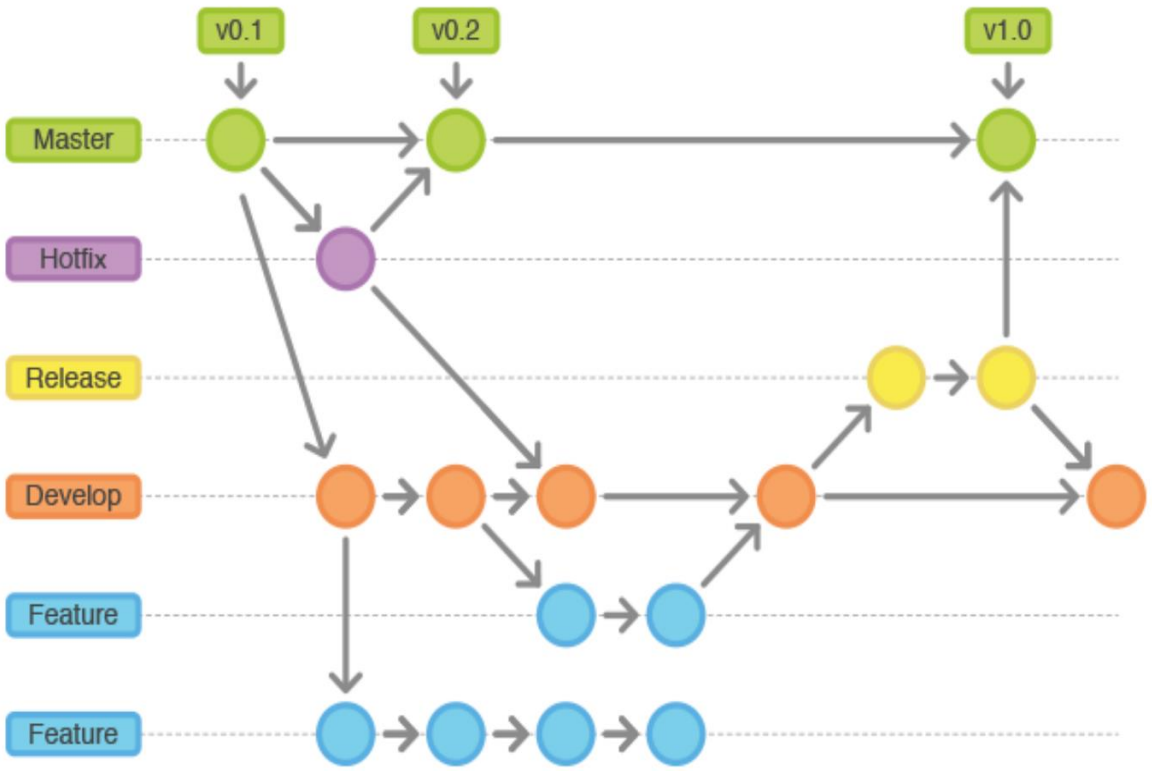
- Share technical approach before implementing.

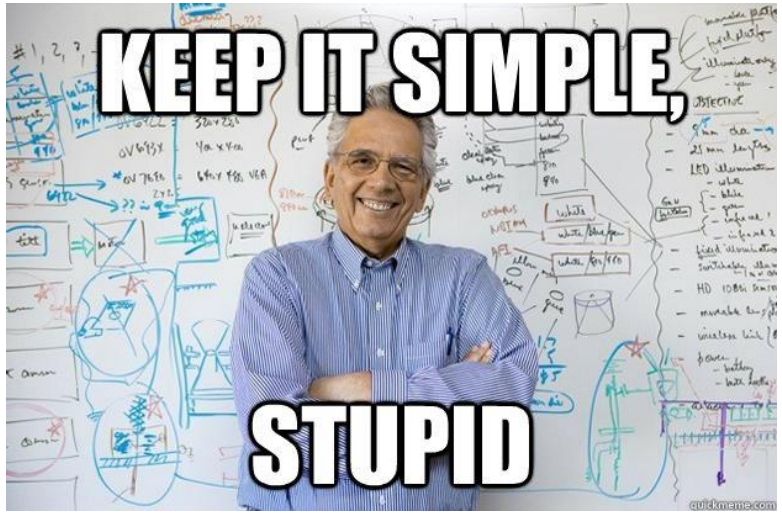
- Add note to Jira tickets or work items
  - Take a picture of technical discussion
  - Enter some words do describe about the technical approach
  - Do not need a big word or pdf document with a lot a diagrams

- Raise issues early

Working with source controls

- Try to understand how git work
- Try to find ways to use git effectively
- **Always preview changes before every commit**





“

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided

https://en.wikipedia.org/wiki/KISS_principle

SEPARATION OF CONCERNS

Don't let your plumbing code pollute your software.

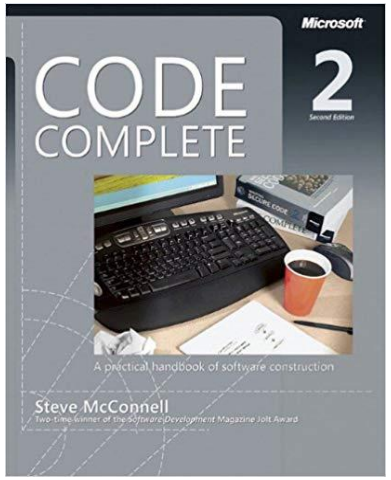
Good developer don’t just write good code

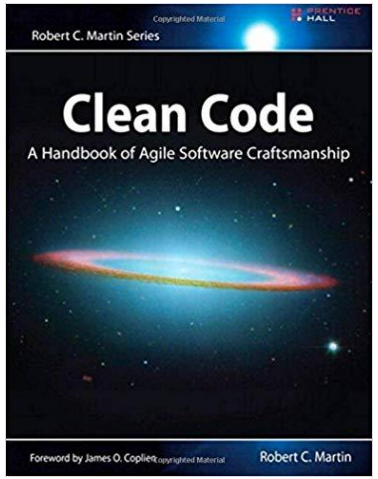
- Good understand the business domain – read requirement carefully
- Good team work
- Get involved
- Continuous improving – continuous learning

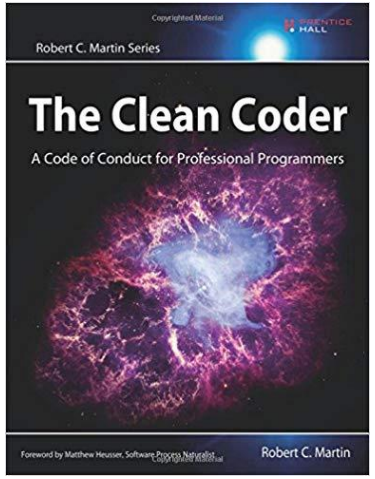
Follow industry coding style and best practice

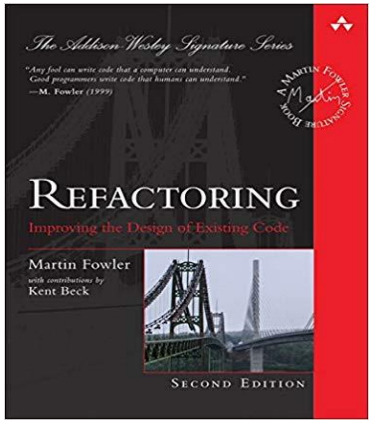
- https://github.com/thangchung/clean-code-dotnet
- https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md
- https://github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AspNetCoreGuidance.md
- https://github.com/airbnb/javascript

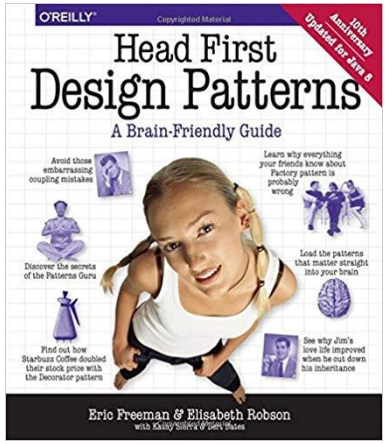
Good books for developers

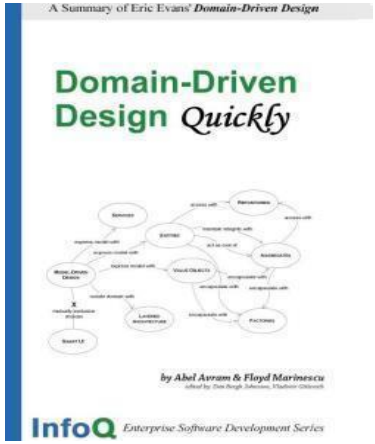












THANK YOU

www.nashtechglobal.com